

“Lean and Efficient Software: Whole-Program Optimization of Executables”

Project Summary Report #2 (Report Period: 12/25/2012 to 3/24/2013)

Date of Publication: April 3, 2013
© GrammaTech, Inc. 2013
Sponsored by Office of Naval Research (ONR)

Contract No. N00014-12-C-0521
Effective Date of Contract: 09/25/2012
Requisition/Purchase Request/Project No.
12PR10102-00 / NAVRIS: 1100136

Technical Monitor: Sukarno Mertoguno (Code: 311)
Contracting Officer: Casey Ross

Submitted by:



Principal Investigator: Dr. David Cok
531 Esty Street
Ithaca, NY 14850-4201
(607) 273-7340 x. 146
dcok@grammatech.com

Contributors:

Dr. David Cok	Dr. Alexey Loginov
Tom Johnson	Brian Alliet
Dr. Suan Yong	David Ciarletta
Dr. Junghee Lim	Frank Adelstein

DISTRIBUTION STATEMENT A: Approved for public release; distribution is unlimited.

Financial Data Contact:

Krisztina Nagy
T: (607) 273-7340 x.117
F: (607) 273-8752
knagy@grammatech.com

Administrative Contact:

Derek Burrows
T: (607) 273-7340 x.113
F: (607) 273-8752
dburrows@grammatech.com

Report Documentation Page			Form Approved OMB No. 0704-0188		
Public reporting burden for the collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to a penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.					
1. REPORT DATE APR 2013		2. REPORT TYPE		3. DATES COVERED 00-12-2012 to 00-03-2013	
4. TITLE AND SUBTITLE Lean and Efficient Software:Whole-Program Optimization of Executables			5a. CONTRACT NUMBER		
			5b. GRANT NUMBER		
			5c. PROGRAM ELEMENT NUMBER		
6. AUTHOR(S)			5d. PROJECT NUMBER		
			5e. TASK NUMBER		
			5f. WORK UNIT NUMBER		
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) GammaTech,531 Esty Street,Ithaca,NY,14850			8. PERFORMING ORGANIZATION REPORT NUMBER		
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)			10. SPONSOR/MONITOR'S ACRONYM(S)		
			11. SPONSOR/MONITOR'S REPORT NUMBER(S)		
12. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited					
13. SUPPLEMENTARY NOTES					
14. ABSTRACT					
15. SUBJECT TERMS					
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT Same as Report (SAR)	18. NUMBER OF PAGES 21	19a. NAME OF RESPONSIBLE PERSON
a. REPORT unclassified	b. ABSTRACT unclassified	c. THIS PAGE unclassified			

1 Financial Summary

Total contract amount (1 year)	\$399,984.00
Costs incurred during the performance period (1/1/2013-3/31/2013)	\$177,882.41
Costs incurred to date (to 3/31/2013)	\$214,135.73
Estimated to complete	\$185,848.27

2 Project Overview

Background:

Current requirements for critical and embedded infrastructures call for significant increases in both the performance and the energy efficiency of computer systems. Needed performance increases cannot be expected to come from Moore's Law, as the speed of a single processor core reached a practical limit at ~4GHz; recent performance advances in microprocessors have come from increasing the number of cores on a single chip. However, to take advantage of multiple cores, software must be highly parallelizable, which is rarely the case. Thus, hardware improvements alone will not provide the desired performance improvements and it is imperative to address software efficiency as well.

Existing software-engineering practices target primarily the productivity of software developers rather than the efficiency of the resulting software. As a result, modern software is rarely written entirely from scratch—rather it is assembled from a number of third-party or “home-grown” components and libraries. These components and libraries are developed to be generic to facilitate reuse by many different clients. Many components and libraries, themselves, integrate additional lower-level components and libraries. Many levels of library interfaces—where some libraries are dynamically linked and some are provided in binary form only—significantly limit opportunities for whole-program compiler optimization. As a result, modern software ends up bloated and inefficient. Code bloat slows application loading, reduces available memory, and makes software less robust and more vulnerable. At the same time, modular architecture, dynamic loading, and the absence of source code for commercial third-party components make it hopeless to expect existing tools (compilers and linkers) to excel at optimizing software at build time.

The opportunity:

The objective of this project is to investigate the feasibility of improving the performance, size, and robustness of binary executables by using static and dynamic binary program analysis techniques to perform whole-program optimization directly on compiled programs. The scope includes analyzing the effectiveness of techniques for specializing library

subroutines, removing redundant argument checking and interface layers, eliminating dead code, and improving computational efficiency. The contractor expects the optimizations to be applied at or immediately prior to deployment of software, allowing them to tailor the optimized software to its target platform. Today, machine-code analysis and binary-rewriting techniques have reached a sufficient maturity level to make whole-program, machine-code optimization feasible. These techniques open avenues for aggressive optimization that benefit from detailed knowledge of an application's composition and its environment.

Work items:

We expect to develop algorithms and heuristics to accomplish the goals stated above. We will embed our work in a prototype tool that will serve as our experimental and testing platform. Because "Lean and Efficient Software: Whole-Program Optimization of Executables" is a rather long title, we will refer to the project as *Layer Collapsing* and the prototype tool as **Laci** (for **LA**yer **C**ollapsing **I**nfrastructure).

The specific work items are listed below:

1. The contractor will investigate techniques for specializing libraries and third-party components—i.e., techniques for deriving custom versions of libraries and components that are optimized for use in a specific context.
 - 1.1. The contractor will evaluate program-slicing and program-specialization technology developed independently at the referenced university.
 - 1.2. The contractor will investigate techniques for recovering intermediate program representation (IR) required for slicing and specialization techniques. The contractor will focus on the following tasks:
 - 1.2.1. Using static binary analyses for IR recovery.
 - 1.2.2. Using hybrid static and dynamic binary analyses for IR recovery.
 - 1.2.3. Studying trade-offs between the two approaches.
 - 1.2.4. Identifying the approach to be implemented in a prototype tool.
2. The contractor will attempt to implement a prototype optimization tool. This objective can be subdivided into the following subtasks:
 - 2.1. Implement IR-recovery mechanisms.
 - 2.2. Extend and improve the implementation of the slicing or specialization technology transferred from the university.
 - 2.3. Investigate the tradeoff between improved performance through specialization and the resulting increase in executable size.
 - 2.4. Investigate options for handling dynamically linked components and libraries.
3. The contractor will investigate techniques for further optimization of executables and for collapsing library interface layers. The contractor will consider:
 - 3.1. Selective inlining of library functions.
 - 3.2. Specialization of executables to the target platform.As time and resources permit, the contractor will attempt to implement these additional techniques in the prototype optimization tool.

4. The contractor will evaluate the prototype optimization tools implemented or received from the university experimentally. The contractor will use synthetic benchmarks, as well as real-world open-source software for the evaluation.
5. The contractor will maintain project documentation and produce comprehensive progress reports and a detailed final report.

3 Staffing

The following personnel are participating in this project.

Dr. David Cok is the PI and is responsible for program management, infrastructure and the user-facing aspects of the resulting tool. He is also the PI for GrammaTech's effort on the DARPA Rapid project; that project is producing some key underlying technology that is being used by the Layer Collapsing project.

Dr. Alexey Loginov is the key architect of the binary analysis infrastructure.

Dr. Suan Yong and **Dr. Junghee Lim** are senior scientists having detailed knowledge of the binary analysis infrastructure and algorithms.

Brian Alliet is the principal implementation engineer.

Tom Johnson is the resident expert on the API for editing the Intermediate Representation of an analyzed binary. He will be consulted regarding the current state and designs for improvement of this API.

David Ciarletta and **Frank Adelstein** are contributing to infrastructure development and measuring overall algorithm and tool robustness.

4 Accomplishments during the reporting period

4.1 Overall plan

The principal goals for the first three months of the project were to plan the details of the project work, to assess the applicability of existing tools and algorithms, and to perform some feasibility experiments.

In the three month period just ending, we

- performed studies of possible useful transformations,
- implemented some of them,
- implemented the infrastructure to test that the transformations do not alter the valid behavior of the transformed programs, and
- continued to review and exercise the specialization implementation from UWisconsin.

The following sections provide details on these accomplishments.

4.2 Transformations

At the heart of the Laci system will be a set of transformations that can be performed on an executable binary program, possibly accompanied by dynamically loaded libraries. Each transformation is expected to preserve all valid functionality of the program, but to provide some benefits. We are measuring benefits in three areas. These three goals may be differently emphasized in different contexts.

- Changes in size of the executable – reductions in size are expected to translate into better use of resources and better efficiency, e.g., due to decreasing the load on the instruction cache
- Changes in runtime performance – better runtime performance is always desired by users, and is also correlated with lower power consumption
- Changes in security vulnerabilities – transformations induce diversification, making the executable harder to exploit; additionally, removing some procedures from the executable reduces the number of return statements, and thus the number of potential ROP gadgets available to attackers.

We address each potential transformation with the following steps:

- **Limit studies:** where possible, before beginning to implement a transformation, estimate how much benefit is reasonable to expect from the transformation. Note that modern compilers implement many very sophisticated optimizations, so some transformations may turn out to yield minimal benefit. It is best to know how much benefit to expect before diving into implementation work; having an estimate also helps to evaluate the success of the implemented transformation.
- **Transformation:** implement the actual transformation on the binary executable (and libraries, if relevant), using the GrammaTech's CodeSurfer Intermediate Representation.
- **Candidate selection:** devise the decision procedure that indicates when to apply the transformation. For example, an inlining transformation is able to inline a procedure at any call site. Candidate selection will decide when to apply the transformation. A reasonable approach may be to inline only those procedures that are called exactly once, producing slight size, performance, and security improvements. However, it may be beneficial to apply inlining to procedures that are called more than once, increasing the application size, but providing more performance and security benefit. Candidate selection will embody algorithms for such decisions.
- **Evaluation on crafted applications:** Validate that the transformation achieves its benefits on subject applications designed especially to demonstrate the algorithm.

- **Evaluation on a test suite:** We are compiling a test suite consisting of a sampling of realistic executables. This evaluation will provide data on how successful the transformation can be expected to be in practice.
- **Threats to validity:** The transformation or the heuristics about when to apply it may intentionally or by necessity exhibit limitations. We intend to document the situations in which the transformation is expected to be sound (i.e., preserve all valid behaviors of the program) and when it may not be. It is possible that a transformation that is not sound on some classes of applications may still have significant benefit.

The final dimension of our study is the set of transformations themselves. We list candidate transformations (or variants) here and discuss them further in the subsections below.

- The NULL transform
- Dead code removal
- Procedure inlining
- Converting dynamically linked functions to statically linked ones (aiming to remove shared objects and dynamically loaded libraries)
- Specialization (possibly including partial evaluation)

4.2.1 The NULL transform

The NULL transform does not perform any changes to the structure of the program. However, the program is analyzed into an Intermediate Representation and then written out again. As a result, the ordering of basic blocks and procedures may change, failures to identify relocatable symbols may result in a faulty end result, among other changes. Thus the NULL transform is a test of the basic Laci infrastructure: if an application still performs correctly after the NULL transform, then the Laci infrastructure is working correctly (or at least correctly enough for that particular application).

Status: The NULL transform is applied as part of nightly tests to the whole test suite and each test application passes all of its regression tests.

4.2.2 Dead code removal

Transformation. The dead code transform removes code from the subject program that is not executed. Dead code can consist of procedures that are never called or basic blocks within a procedure to which control is never transferred. Closely related to dead code removal is the elimination of data areas that are never used.

Our first target is entire unused procedures. The transformation to eliminate these is straightforward, since it just consists of eliminating the procedure – no other code needs adjusting (besides the usual relocation of code that happens when object modules are assembled into a working executable).

The actual transformation does more than just check whether a procedure is called. Rather, this transformation removes code that is statically known not to be reachable via the control-flow graph from the entry point (usually `main`). Given an object file with three procedures, only one of which is used, linkers will nearly always link the two unused procedures, simply because they are in the same object file (object files are generally treated as an atomic unit). With the control-flow information provided by our binary analysis, we can determine that these extra procedures aren't reachable, and remove them. The dead code removal transformation works similarly to a garbage collection algorithm. It initially marks the entry point (e.g., `main`) as reachable, then continues traversing the control-flow graph, recursively marking procedures as reachable. When the algorithm terminates, any node left unmarked is dead and can be removed. Note that data references have to be taken into account as well. For example, `mov eax, proc1` makes `proc1` reachable, even if control-flow analysis may miss indirect calls to `proc1` enabled by the instruction.

Applying the transformation. Knowing when to apply the transformation is a bit trickier. One can readily determine whether there are any direct calls to a procedure from elsewhere in the program. Note that there may be groups of procedures that mutually call each other but no procedure in the group is called, and the group as a whole is dead.

However, procedures can also be called indirectly. Knowing the values of all function pointers requires a much more complicated analysis; in general, it is undecidable. A conservative approach can note when the address of a procedure is taken, even if it cannot determine whether that address is ever used. However, even then, a particularly obfuscated program could construct function pointers from integer operations. Thus there is some soundness risk in applying this transformation.

Benefits. This transformation is expected to significantly reduce the size of an executable and improve security by reducing the number of ROP gadgets. No direct effect on runtime performance is expected; some speedup may result because there is less code to read from memory when the program starts and instruction-cache locality may be improved.

Limit study. To determine how much value may be obtained by dead code removal we performed the following study. Static and dynamic analysis data was collected to gather statistics on the frequency of calls for each procedure in the test executables. A total of 104 executables from the GNU coreutils package are regularly analyzed by the test infrastructure against completed transformations (currently the NULL transform). The analysis gathers information about procedures in each executable to identify if they have formal arguments, if there are local variables, how many places the procedure is called from, and if any of those call sites use constants as one or more of the arguments. **Error! Reference source not found.** shows a histogram of the number of procedures with X call sites (where X is the value on the x-axis) across all of the coreutils executables. A total of 5540 thunks are excluded from this data set. Thunks (small delegating procedures) make up about 1/3 of the total procedures across all executables analyzed and will be removed when the executable is rewritten.

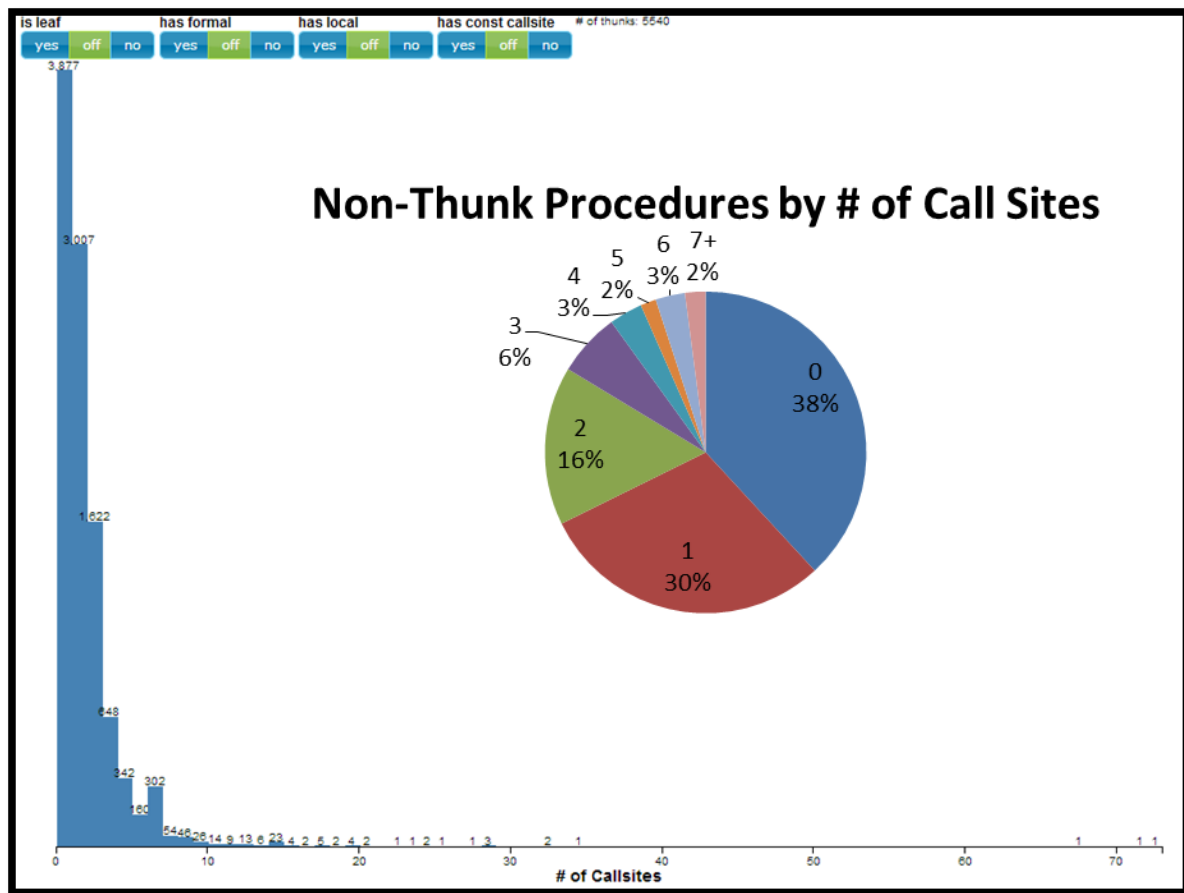


Figure 1 - Total Procedures by # of Call Sites for GNU coreutils

Of immediate note is the number of procedures with no call sites (3,877, 38.1%). This is largely due to the use of shared objects among the executables in the coreutils package. When any procedure from one of those shared objects is referenced, the entire object is linked into the executable.

Results. The dead-code-removal transformation was applied to our test executables. The results are shown in the following graph.

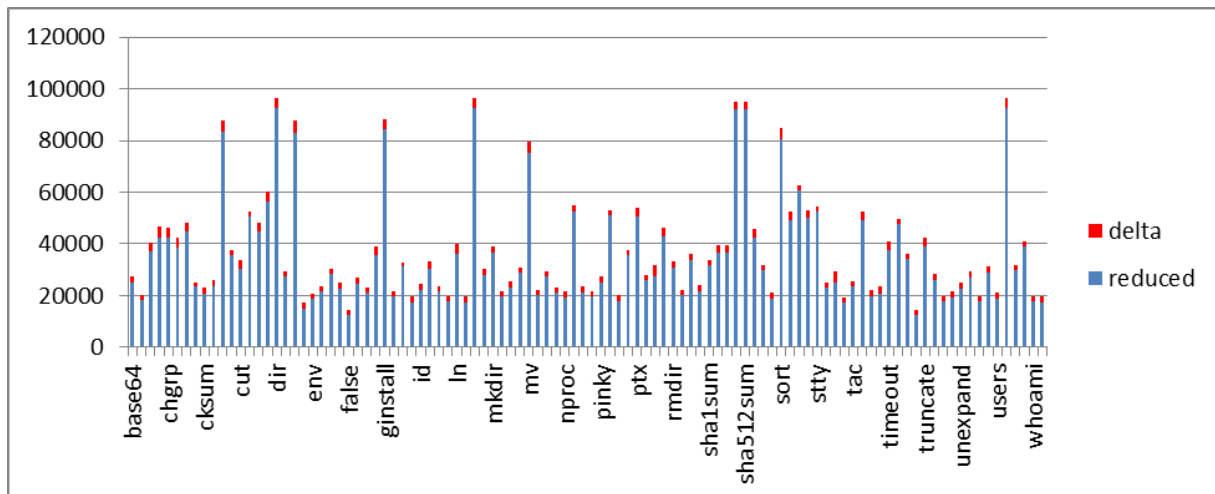


Figure 2 - Reduction in executable size from dead-code removal

The net result on this data set is that the transformation reduced the size of executables by 4-12%, with an average of 7-8%. We also counted the reduction in the number of procedures, shown in Figure 3.

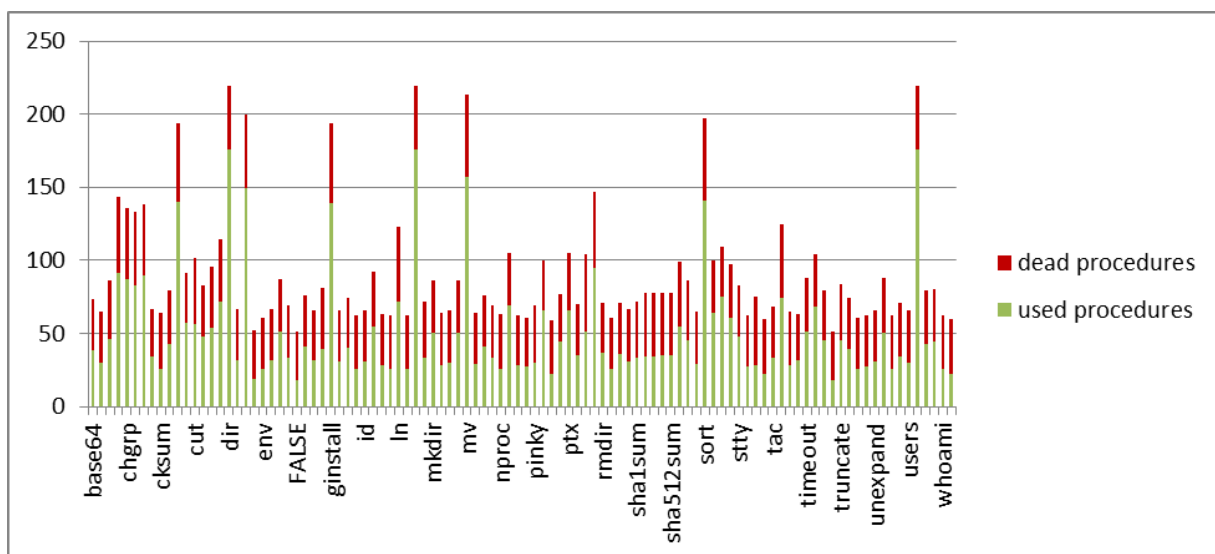


Figure 3 - Reduction in the number of procedures from dead-code removal

The reduction in number of procedures ranged from 20-65%, with an average of about 47%. The procedures removed are on average smaller than the average procedure in the applications.

4.2.3 Inlining procedures

Transformation. The inlining transformation replaces a call to a procedure with the actual text of the procedure, removing control transfer (usually effected by `call` or `jmp` instructions). The complete application of the transformation can be carried out using the

following intermediate steps (not all of these steps are necessary for improved performance, power consumption, and security):

- Replace the call to the callee and return to the caller with jump instructions.
- Remove the jump instructions and reorder the code so that the instructions of the callee are directly between their immediately preceding and following instructions in the caller.
- Combine the activation records of the two procedures so that the stack size is adjusted just once to account for the callee's local variables and saved registers. Adjust the stack to account for the removal of some operations, such as the saving of the return address.
- Remove what used to be the copies of values to stack locations that held the actual arguments of the callee. Change the callee to use the values directly from the caller's activation record (or as global memory locations). This can be accomplished by analyses such as copy propagation and constant folding.
- Adjust the stack to avoid saving space for the return address.
- Eliminate unnecessary register-save operations, e.g., registers saved by the caller that are not modified by the callee.

4.2.3.1 Inlining procedures without cloning

Applying the transformation. This transformation can be applied whenever a procedure is called just once and its address is never taken. Some of the same caveats apply as for the analysis of calls to determine dead code.

Benefits. This transformation is expected to decrease the size of an executable and increase its runtime performance only slightly. (The improvement comes from avoiding unnecessary parameter marshaling and control transfers.) However, the transformation will reduce the number of return instructions and therefore improve security by reducing the number of ROP gadgets.

Limit study. The limit study results for dead code above also contain information about inlining possibilities. In the data set we analyzed, 3007 (29.5%) of the procedures had only one call site and are reasonable candidates for inlining.

Results. This transformation has been successfully implemented. The number of procedures removed equals the number of RET instructions removed, and is a measure of the security improvement.

4.2.3.2 Inlining procedures with cloning

Transformation. Inlining can be applied even if a procedure is used in more than one place: a separate copy of the procedure can be inlined in each location where it is used.

Applying the transformation. Just as for inlining without cloning, this transformation can be applied wherever the callers of a procedure are precisely known and its address is never

taken and used. The challenge is to balance the size of the inlined procedure and the number of inlined clones against the resultant increase in executable size.

Benefits. This transformation is expected to increase the size of the executable and decrease the number of ROP gadgets. The direct effect on performance will stem from reducing parameter passing and control transfers.

Results. We have not yet implemented the cloning operation.

4.2.4 Converting dynamically linked functions to statically linked

Benefits. Many library routines are made available to executables as shared objects or as dynamically loaded libraries. The advantage of this system design is that, when multiple executables use the same routines, the system need load those routines into memory only once. If each routine were linked statically into each executable, the sizes of all of the executables would increase significantly. However, the presence of shared objects means that there are also many procedures that are part of the address space of an executable but are not used by it; thus there is dead code and an increased number of possible ROP gadgets.

Transformation. The transformation is to statically link in only the needed functions from a dynamically loaded library module, removing the need to use shared objects and DLLs. This requires essentially reimplementing (or reusing) much of the functionality of the linker and loader.

Applying the transformation. The transformation can be applied whenever the analysis can be confident that it knows which of the procedures in a shared object or DLL are called and at what call site. The benefit of the transformation to the transformed application is easy to understand. However, real-world applications of the transformation need to take into account the effect on the whole system: will other applications run more slowly because the extra copies of previously shared code affect the virtual memory and the instruction cache. Distinguishing standard from application-specific libraries may provide an acceptable answer.

Limit study. The limit study above identified a large fraction (38%) of procedures that were unused; most of these are from shared objects.

Results. Implementation of this transformation is expected to begin in the next performance period.

4.2.5 Specialization and partial evaluation

Specialization and partial evaluation are terms that are often used to refer to similar techniques (we will adhere to the common practice of viewing partial evaluation as a kind of specialization, as will be elaborated below). The primary goal of specialization is to improve the runtime cost of a program by optimizing the program's code for the restricted context in which the program components (e.g., functions) are used. These are some example use cases:

- A program performs many computations but only some of those computations affect the program outputs (e.g., some command-line processing has no effect on the output or observable side effects). Instructions that do not affect the outputs can be removed.
- General purpose library routines may be used in just a few contexts within a program. As a result, some of the instructions within the library routine may be unused and removable. For example, perhaps a procedure tests that a given argument is non-null, issuing an error message if it is null. If analysis shows that in all of the calling contexts in a subject application the caller assures that the argument is indeed non-null, then the error-checking code within the library routine can be removed.
- Values known at load time (e.g., what kind of a platform a general library routine is being executed on) may enable the more general kind of specialization when instructions are *partially evaluated* to produce simpler or more efficient specialized code.

These transformations can be implemented with varying degrees of complexity.

- A dependence-based specialization considers only the data and control dependences within a program, given its limited context of use, and does not try to concretely or symbolically evaluate the program. This can provide a conservative amount of code reduction. This is implemented in UW's specialization-slicing prototype.
- Partial evaluation usually refers to specialization that involves simplifying code based on information known statically. For instance, constant propagation can be used to simplify uses of a procedure's argument that is known statically. Partial evaluation begins with binding-time analysis that determines the division of arguments to a function into those known statically and those determined only at runtime.
- Polyvariant specialization (or partial evaluation) is capable of creating multiple copies of a procedure specialized to multiple distinct contexts (i.e., different constant values for a given argument known statically). Transformations of copies can be aggressive, as they can be tailored to a single context. However, the gain needs to be balanced against the indirect costs of creating many similar forms of a procedure.
- Polyvariant-division specialization (or partial evaluation) is prepared to consider multiple divisions of arguments into statically known and dynamically determined. This requires the ability to create multiple copies of the procedure to be specialized. Prof. Reps's group at UW Madison intends to work with GrammaTech to extend the specialization-slicing prototype into a polyvariant and polyvariant-division partial-evaluation tool.

Limit study. One initial study of the possibilities of specialization was to observe in our data set how frequently the arguments to procedures are constants and which constant values were used. These locations would be potential targets for specialization via partial evaluation. Figure 4 displays the histogram filtered for procedures with at least one call site that uses a constant argument. By selecting one of the bars in the histogram we can then drill down into the data to see details on each call site and determine the number of specialized

copies that would be required. For example, procedure `gnu_mbswidth` in the `dir` executable is called 6 times with an explicit zero passed as the second argument. This provides an example of a prime target for specialization via partial evaluation because the original procedure could be replaced by one specialized copy.

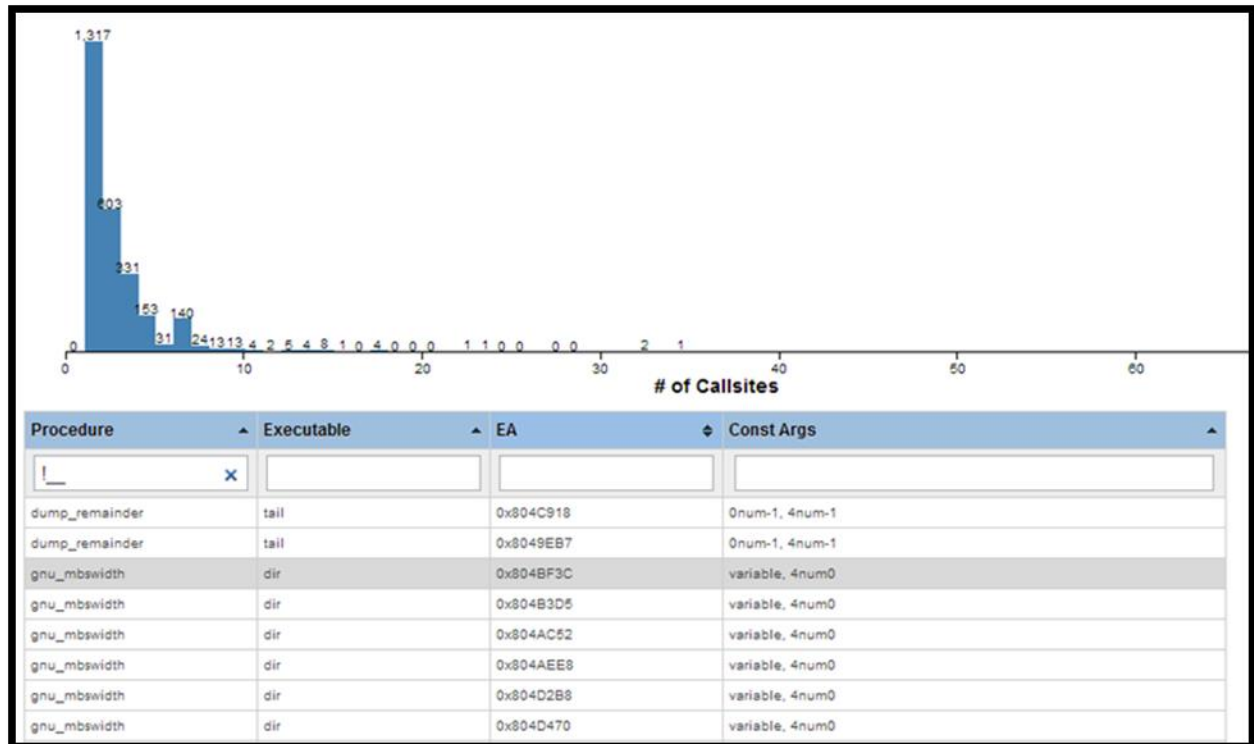


Figure 4 – Call Site Histogram and Detail Table Filtered for Procedures with a Call Site with a Constant Argument

Status. This transformation is under study and has not yet been implemented.

4.3 Evaluation of specialization slicing

During this reporting period we evaluated both the ideas and the implementation behind the work on Specialization Slicing done by our collaborators at the University of Wisconsin¹.

The main contribution of that work is an algorithm for performing polyvariant specialization. The technique has a reasonably efficient polynomial cost *in practice* (while in theory there is a possibility of exponential explosion). The specialization is based on slicing, i.e., by following

¹ Aung, M., Horwitz, S., Joiner, R., and Reps, T., Specialization slicing. TR-1776, Computer Sciences Department, University of Wisconsin, Madison, WI, October 2012. Submitted for journal publication.

dependence edges only, and does not simplify code based on the values of constants and variables. There may be benefit to combining this approach with a partial-evaluation-based approach that considers values. UW and GrammaTech intend to pursue this further.

The prototype implementation targets C programs, using CodeSurfer/C as a front end. We have imported a copy of Wisconsin's implementation into our code repository, and are working to reproduce the results presented in their paper. Thereafter, we'll consider several possible tasks:

1. Improving the performance of the prototype, which at times was limited by the specific version of CodeSurfer available to UW. Given that we have full access to the CodeSurfer source code, we can make improvements within CodeSurfer to support operations needed by this work.
2. Adapt the work to target x86 binaries. Given that the work uses the CodeSurfer framework, it already leverages some existing common infrastructure that supports both C and x86 programs. However, peculiarities of analyzing binaries must be accounted for, like safely excluding dependence edges induced by updates to the stack pointer, which would normally (in a naïve slicing implementation) result in poor quality slices (slices that are too big, effectively including the entire program).

4.4 Rewriting infrastructure

The program transformations that we are implementing as part of Laci are built on a foundation of rewriting operations on the basic Intermediate Representation. For the purpose of Laci, we need additional rewriting operations – in particular deletion operations. Such operations are non-trivial because there is a great amount of interdependent analysis results and data tables; changing the IR requires appropriately updating all related data (or recomputing it from scratch).

In this reporting period, we improved the pretty printer and identified and fixed IR problems that caused rewriting to fail. Over a dozen different types of IR problems were identified and test cases were created for them. A mechanism for providing "user hints", that is assembly-IR hints provided by the user in an external human readable file, was created to allow us to quickly work around the problems, and identify exactly what extra information or computation is needed to overcome them. Hints were created for all the microtests for the IR problems identified, as well as all of coreutils, allowing them all to rewrite successfully and pass the regression tests.

Using the number and frequency of the hints required for the various tests, we identified the IR problems that were most prevalent and began to address them. The most prevalent problem was misidentifying operands as numeric instead of symbolic (e.g., `lea eax, [eax*4+0x1234]`, where `0x1234` is an address in the data segment). We developed a heuristic to identify these cases (which while not entirely sound, is justified by all our experimentation to date) and correct the IR. A handful of other more minor IR problems

were also fixed (including data vs. code confusion, jump tables misidentified, and shared-library imports misidentified).

Other accomplishments include optimizing instruction selection for instructions with multiple encodings and support for removing code (identified as dead) from our IR.

Next, we give more details on improvements to the rewriting infrastructure carried out during this performance period.

The virtual memory addresses of code and data in transformed binaries are nearly always different from their initial values. This can simply be due to reordering existing code (which may happen as a side effect of the machine code to CodeSurfer IR translation, or intentionally to improve security), or due to the removal of dead code and data from the binary. When code and data moves, all references to their old locations must be updated to reference the new locations. This is accomplished by making memory references *symbolic* in the IR. For example: `mov eax, [0x1234]` might be represented as `mov eax, [global_var_1]` in the IR. Which indicates that we want to load the value of the `global_var_1`, wherever it is placed, not necessarily what is at address `0x1234`.

Unfortunately identifying references which should be symbolic is difficult. In the above example it is relatively straightforward because the instruction explicitly dereferences `0x1234` (accessing the data at that memory location). However, if instead the instruction were `mov eax, 0x1234`, it would be generally impossible to decide whether it's an address (demanding a symbol) or a literal value without additional information. This is because the compiler could generate such an instruction for C statement `void *p = &global_var_1`, as well as `int x = 0x1234`. We developed heuristics that intend to distinguish between these two cases and mark operands as symbolic where desired. The heuristics take into account a variety of potential hints including the range of valid addresses in the binary, the alignment of the value (e.g., pointers in the data section are always four-byte aligned), and the distance from known symbols (e.g., information gathered from the symbol table, if provided). According to our evaluation so far, these heuristics work remarkably well in practice.

The x86 ISA has multiple encodings (which are functionally equivalent, but are represented by different byte sequences) for some instructions. For example, `mov eax, 42` can be encoded using a four-byte immediate for the literal value `42`, or a one-byte immediate. They both accomplish the same thing, but the former takes three more bytes to encode. Our intermediate representation did not previously distinguish between these encodings for simplicity. However, when transforming a binary, it is important not to use less optimal encodings. Since some of this information was previously lost in the IR, our code generator had to redo some of these optimizations to avoid creating larger binaries. We added support for a handful of alternate encodings (preferring the optimal size for immediate values) to our code generator, and with additional IR support we can now emit byte-for-byte—modulo code reordering—identical binaries when no other transformations are performed.

The dead code removal transformation described earlier motivated the development of support for removing complete procedures from our IR. While dead code removal could be implemented (and indeed, initially was) as a filter on the code to be emitted following transformation, it is more naturally implemented as a modification of the IR. This means that the code generator no longer has to be aware of the dead code removal pass, and later transformations can take advantage of the results of dead code removal (for example, a procedure with two call sites, one of which is dead, can now be subject to inlining if the inlining transformation doesn't see the dead call site). We have implemented support for removing arbitrary procedures (as well as individual instructions or basic blocks within procedures) from our IR. These removals must be done in a way that doesn't invalidate the CFG. For example, a procedure can only be removed if it has no callers. The implementation ensures that removing a procedure is a sound transformation.

4.5 Evaluation infrastructure

An important aspect of developing a successful Laci prototype is to establish a common evaluation approach that can track the success and performance gains of applying optimizations to an executable. In January, a testing and performance evaluation framework was built to establish a baseline and track progress on the development of the prototype. The test infrastructure supports submission of test cases that specify transformations to apply to target executables. Once submitted, the test cases are run on a daily basis and test results are recorded using existing GrammaTech test infrastructure. The current tests apply a transformation to an executable, performing optional IR validation if specified. If the transformation is successfully applied (i.e., a new executable is written), then user-defined tests are run against each version of the executable (pre- and post- transformation). Any discrepancies in test results are reported as a failure. Additionally, system-resource utilization metrics are collected for each test run on each version of the executable and average changes in utilization profiles are reported. These metrics will be expanded as project objectives are refined by initial investigations, but establish a solid baseline upon which general gains in performance, size and security can be tracked as transformations are developed. The test infrastructure is currently being populated with both small sample programs that test the soundness of the IR being generated and with a set of real-world executables that currently consists of the GNU coreutils package. As transformations are developed they can be tested against these and other executables added to the infrastructure.

The test and performance evaluation framework was expanded in February to support limit studies to guide the prioritization of transformations to prototype, as described above. In March, we identified additional applications to add to the test and performance evaluation framework including: bzip2, cryptopp, glut, libpng, and Python. These applications and their test suites will be used to enhance the validation of our prototype transformations and extend our limit studies to larger user applications.

Thus, currently, we have a regression suite consisting of

- More than 100 coreutils programs and their regression suites
- Additional 3rd party software. We are evaluating bzip2, cryptopp, glut, libidn, libpng, minipnp, Python, sphinx, yasm, and zlib. We have incorporated bzip2 into Laci and are in the process of incorporating libpng. cryptopp is a windows-only package, and thus is currently excluded. The glut package is 15 years old and has linking problems, possibly depending on old libraries. After the other packages have been evaluated we may return to it if more are needed. The remaining packages will be evaluated and added subsequently. [Note that our focus is on Linux executables.]

We are adding additional programs as we find suitable candidates. The key criteria for including a program are the availability of (1) openly available source so that we can build and test it in a variety of environments and (2) a reasonably strong test suite (because it would be a substantial tax on this project to create such test suites ourselves).

We are investigating the use of automated test-case generation via *concolic* execution as implemented in GrammaTech's Grace research tool. The application of Grace to coreutils packages is part of a different contract at GrammaTech. Research carried out under that contract may be able to provide a comparison of the quality of coverage of coreutils test suite with that of our concolic engine Grace. If Grace provides deeper coverage, we may choose to use it for validating our transformations. If Laci includes aggressive transformations that are not always safe, the project may choose to rely on automated test-case generation, as the availability of comprehensive test suites cannot be expected of all applications to be transformed. Concolic execution is a promising approach to this problem.

5 Goals for the next reporting period

In the next reporting period we expect to begin or complete the following (see the milestones table for dates):

- Begin the dynamic to static procedure transformation.
- Complete the evaluation of the UW technology.
- Complete the investigation of procedure inlining.
- Continue the assessment of specialization and partial evaluation technologies and implementations, converting relevant software to our use.
- Continue to add benchmarks and associated test cases to the testing suite

6 Milestones

Interim results on multi-month tasks will be reported in the quarterly progress reports.

Milestone	Planned Start date	Planned Delivery/ Completion Date	Actual Delivery/ Completion Date
Kickoff meeting		As scheduled by Technical Monitor	Phone discussion in January 2013; TM declined to schedule a more in-depth discussion
Evaluation of structure and code quality of UW technology (task 1.1)	10/2012	11/30/2012	11/30/2012
First Quarterly report (task 5)		1/3/2013	1/7/2013
Investigate and implement dead-code removal of entire functions(task 3)	12/2012	3/31/2013	3/2013
Implement a testable working prototype with the null-transform option (the foundation for tasks 2 and 4)	12/2013	2/28/2013	2/2013
Continuing task: Identify failures resulting from incorrect IR; correspondingly improve or repair the IR recovery techniques. (tasks 1.2 and 2.1)	12/2012	9/24/2013, with all individual improvements noted in quarterly reports	Ongoing task with continuous improvements
Identify common coding idioms and compiler transformations that result in incorrect disassembly (task 2.1)	1/2012	2/15/2012	2/28/2012, with additional improvements as opportunities are identified
Implement a testing infrastructure (task 2.3 and task 4)	1/2013	2/28/2012	2/2013
Design and implement the IR editing infrastructure (task 2).	1/2013	4/30/2013	

Evaluation of performance and precision of UW technology (task 1.1)	2/2013	3/31/2013	Still in progress
Develop real-world and synthetic benchmarks to evaluate performance (task 4).	2/2013	9/24/2013, with interim progress each month	2/2013 (infrastructure in place; adding additional tests is an ongoing task)
Investigate disassembly improvements such as learning-based bottom-up disassembly and all-leads disassembly (task 2.1)	3/2013	5/31/2013	
Investigate selective inlining of library functions (task 3.1)	3/2013	7/31/2013	
Second quarterly report (task 5)		4/3/2013	4/3/2013
Investigate finding and deleting functionally dead code, possibly using slicing and specialization (task 2.2 and 3.2).	4/2013	8/31/2013	
Investigate specialization to target platforms or target environments (task 3.2)	4/2013	8/31/2013	
Implement aspects of the chosen disassembly extensions (task 2.1)	5/2013	8/31/2013	
Evaluate hybrid analyses as a complement to static analyses for recovering IR (Task 1.2)	5/2013	8/31/2013	
Third quarterly report (task 5)		7/3/2013	
Measure the performance tradeoff of various optimizations and evaluate the overall tool (task 2.3 and 4)	7/2013	9/24/2013	

Investigate options for handling DLLs (task 2.4)	8/2013	9/24/2013	
Final report (task 5)		10/24/2012 (contract end date)	

7 Issues requiring Government attention

The request to change PI on the project (from David Melski to David Cok) was verbally approved (in January), but the corresponding contracting paperwork has not been received by GrammaTech.